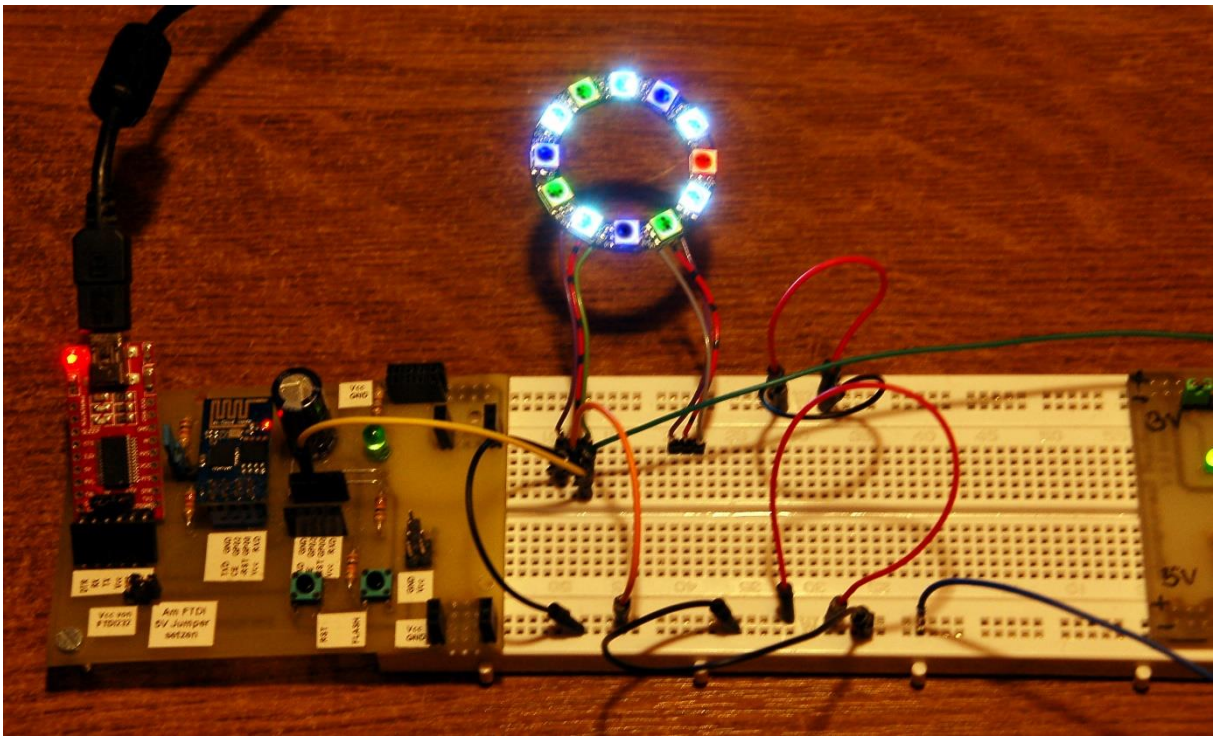

Abbildung 1: TVSIM-Ein TV-Simulator

Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

Ob ein ungebetener Gast, der gesteigertes Interesse am Inhalt Ihrer Wohnung hat (aka Einbrecher), sich davon abschrecken lässt, ist sicher auch von dem Feintuning der folgenden Schaltung und deren Programmierung abhängig. Es gibt die Dinger, von deren Art ich Ihnen heute ein eigenes Design vorstellen möchte, natürlich auch im Handel. Interessant ist es allemal, zu entdecken, was in einer solchen Kiste steckt und wie sie arbeitet. Gemeint sind Geräte, die ein laufendes Fernsehgerät vortäuschen. Wohl denn, willkommen zum Projekt

TV-Simulator

Die Hardware ist schnell zusammengestellt, es sind nicht viele Teile. Und das Programm ist, dank einiger Tricks aus der MicroPython-Schatzkiste relativ kurz. Es umfasst, die paar Kommentarzeilen abgezogen, weniger als 280 Programmzeilen. Dabei dürfen Sie knapp 200 Zeilen für die vorgeschaltete Declaration der Klasse TVSIM abziehen, die sich natürlich auch in eine externe Datei auslagern ließe.

Hardware

Beginnen wir mit der Vorstellung der Hardware. Durch die sehr wenigen Bauteile ist die Schaltung für Anfänger besonders interessant.

1	ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F oder D1 Board NodeMCU ESP8266MOD-12F WiFi ja sogar esp8266 ESP-01S WLAN WiFi Modul Alle diese ESP-Boards und einige weitere sind verwendbar.
1	Battery Expansion Shield 18650 V3 inkl. USB Kabel
1	Li-Akku Typ 18650
1	LED Ring 5V RGB WS2812B 12-Bit 37mm oder ähnlich
1	Breadboard und diverse Jumperleitungen

Wie Sie sehen, sind diverse ESP-Boards für dieses Projekt brauchbar. Und das, mit einer einzigen Ausnahme, ohne Änderung am Programm. Nur der ganz kleine ESP-01 braucht, weil er halt nur GPIO0 und GPIO2 als Portleitungen besitzt, eine einzige Programmänderung in der Zeile, in der die Leitung zum Datenanschluss des Neopixelrings festgelegt wird.

Neopixelringe enthext

Auf dem LED-Ring sind 12 Neopixel-LEDs vom Typ WS2812B verbaut. Die Spannungsversorgung erfolgt parallel. Die Datenleitung führt seriell von einer LED-Einheit zur nächsten und stellt eine besondere Art von Bus dar. Jede Einheit enthält eine RGB-LED und einen Controller, der auf die erste ankommende 24-Bit-Folge der Farbinformation reagiert. Die Signale, mit derselben Periodendauer aber unterschiedlichem Duty Cycle, werden von einem Microcontroller, wie dem ESP32 erzeugt. Je Neopixel-Einheit werden 24 Bit generiert (jeweils 8 für grün, rot und blau). Die Periodendauer für ein Bit ist 1,25µs +/-0,150µs, die Übertragungsfrequenz beträgt somit ca. 800kHz. Für eine 1 liegt die Leitung 0,8µs auf HIGH und 0,45µs auf LOW, eine 0 wird durch 0,4µs HIGH und 0,85µs LOW codiert. Die ersten ankommenden 24 Bits verarbeitet jede WS2812B-Einheit selbst, ohne sie weiterzugeben. Alle nun folgenden werden verstärkt und an die nächste Einheit weitergereicht. Die Signalfolge vom Microcontroller wird also von LED zu LED um 24 Bit kürzer. Anders als bei einem üblichen Datenbus erhalten die WS2812B-Einheiten die Daten aber nicht gleichzeitig, sondern zeitversetzt um jeweils die Dauer von 24Bit mal 1,25µs/Bit = 30µs. Diese Signalfolge wird im ESP32/ESP8266 durch die in MicroPython eingebaute Klasse machine.NeoPixel erzeugt. Die Ansteuerung der LEDs gestaltet sich dadurch sehr einfach, wodurch sich die Anwendung gerade für Anfänger eignet.

Ein Framebuffer (aka Zwischenspeicher) im RAM-Speicher des ESP-Chips speichert die Farbwerte (3 x 256 = 16,7 Mio.) zwischen, und der Befehl NeoPixel.write() schickt die Informationen über den "Bus", der an einem GPIO-Ausgang hängt (bei uns GPIO13), an den Ring. Das ist auch schon alles. Pro Farbe, rot, grün und blau, lassen sich 256 Helligkeitswerte einstellen.

Mehrere Ringe kann man genau so wie einzelne LEDs cascadieren, indem man den Eingang des nächsten Rings mit dem Ausgang des Vorgängers verbindet. Die Anschlüsse erfolgen rückseitig, am besten mittels dünner Litzen.

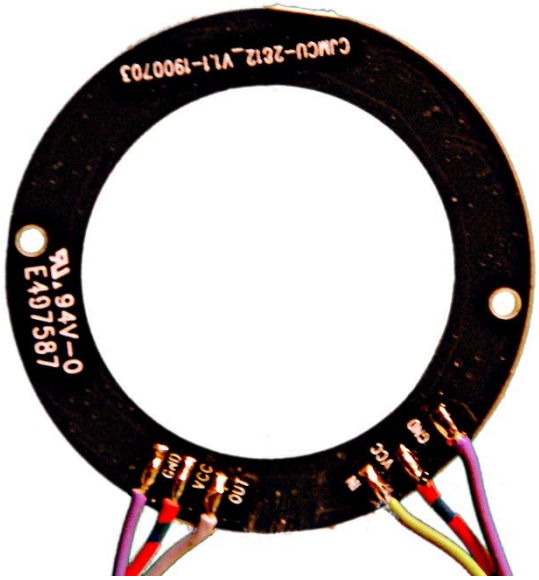

Abbildung 2: LED-Ring_hinten – rechts Zuführung, links Weiterleitung

Die Komponenten für Mischfarben ermittelt man am einfachsten experimentell über REPL. Die Helligkeit der einzelnen Teil-LEDs einer Einheit ist recht unterschiedlich. Die RGB-Farbcodes in den Tupeln werden also bei den Mischfarben selten den gleichen Wert haben.

```
>>> from neopixel import NeoPixel
>>> neoPin=Pin(13)
>>> neoCnt=12
>>> np=NeoPixel(neoPin,neoCnt)
>>> np[0]=(32,16,0)
>>> np.write()
```

Zum Abgleich werden die beiden letzten Befehle mit anderem RGB-Code wiederholt, bis die Farbwiedergabe passt. Die hier angegebenen Werte erzeugen gelb als Mischfarbe von rot und grün.

Mischfarben benötigen wir in diesem Projekt aber nicht. Diese entstehen durch Reflexion des monochromen Lichts aus den einzelnen LED-Einheiten durch die Reflexion an einer weißen Wand, gegen welche die Leuchtseite des Rings gerichtet wird, ganz von selbst.

Abbildung 3: Neopixel-Ring Oberseite

Die Gesamtstromaufnahme des Rings wurde mit einem DVM (aka Digital-Voltmeter) zu ca. 200 mA gemessen. Vermutlich liegen die Stromspitzen bei voller Ausleuchtung bedeutend höher. Der USB-Abschluss reicht für die Versorgung über den Vin-Anschluss des ESP32 jedenfalls nicht aus. Das macht eine extra Stromversorgung, wie die im Schaltschema eingezeichnete Batteriehalterung mit einer Li-Akku-Zelle nötig.

Falls Sie statt des Batteriehalters und des Li-Akkus ein 5V-Netzteil verwenden wollen, müssen Sie die 5V an den Pin 20, Vin, des ESP32 legen.

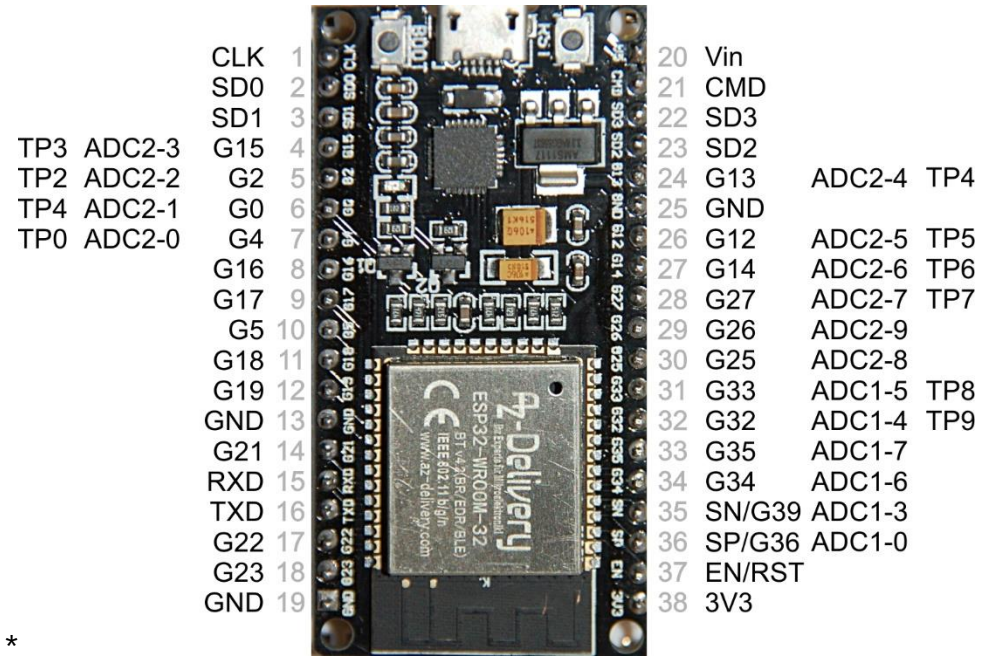

Abbildung 4: ESP32-DEVKITC_V3_Pinout

Die Versorgung aus einem 4,5V-Block aus Alkalizellen ist ebenfalls brauchbar. Legen Sie die +4,5V ebenfalls an den Pin20 des ESP32. Für Versorgungsspannungen über 5V muss ein extra 5V-Regler verwendet werden, denn der Neopixelring darf nicht mehr als 5,3V abbekommen. Für Boards, wie das ESP-01, muss eine extra 3,3V-Versorgung zur Verfügung stehen, denn der kleine Bruder der ESP-Chips hat keinen eigenen 3,3V-Regler. Für eines der ESP8266-Boards habe ich hier noch eine Beschaltungsskizze. Hier können Sie die 5,0V oder 4,5V an Vin legen. Der Minuspol der Versorgung geht an einen GND-Anschluss.

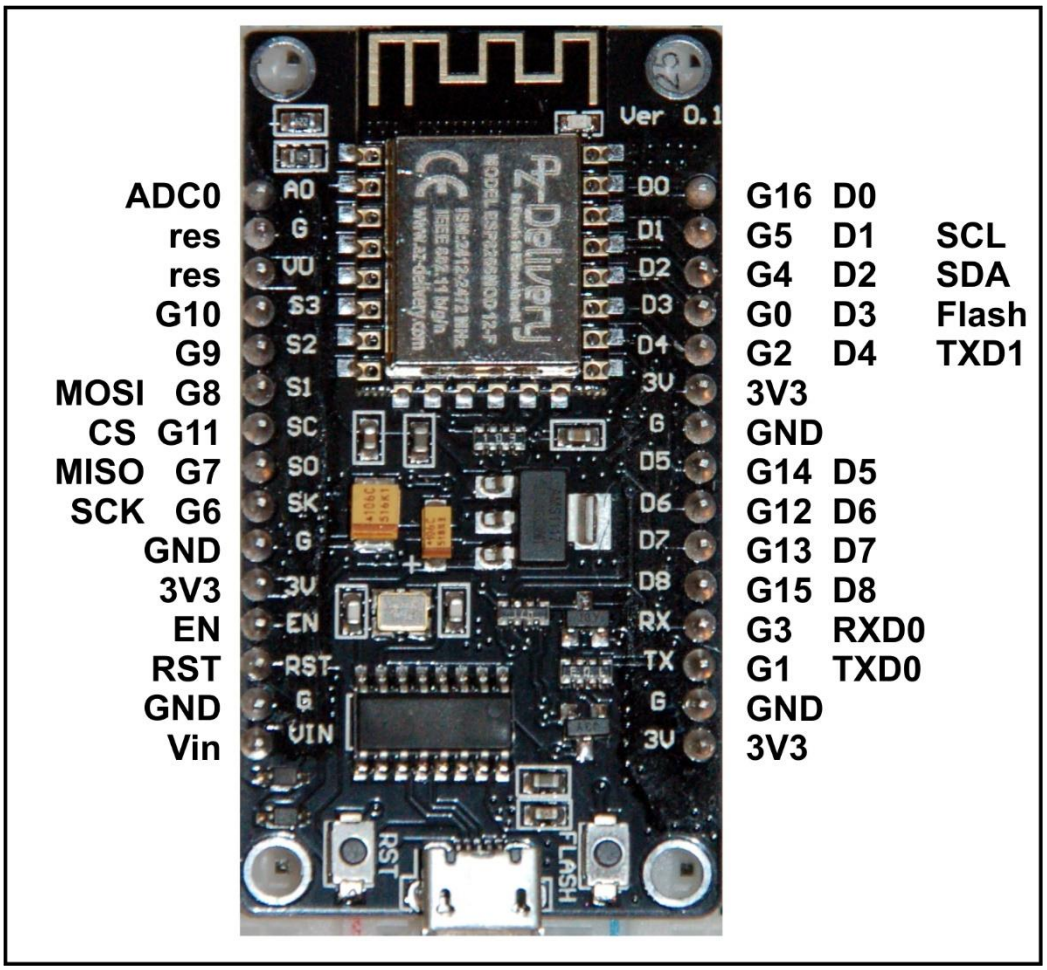

Abbildung 5: ESP8266-NodeMCU-V3

Zum Experimentieren eignen sich übrigens alte PC-Netzteile sehr gut, weil sie neben 5V auch 3,3V und 12V zur Verfügung stellen. Damit können auch hungrige Stromfresser zufriedengestellt werden.

Die folgende Abbildung zeigt das Schaltschema. Ein besser lesbares Exemplar in DIN A4 können Sie als PDF-Datei downloaden.

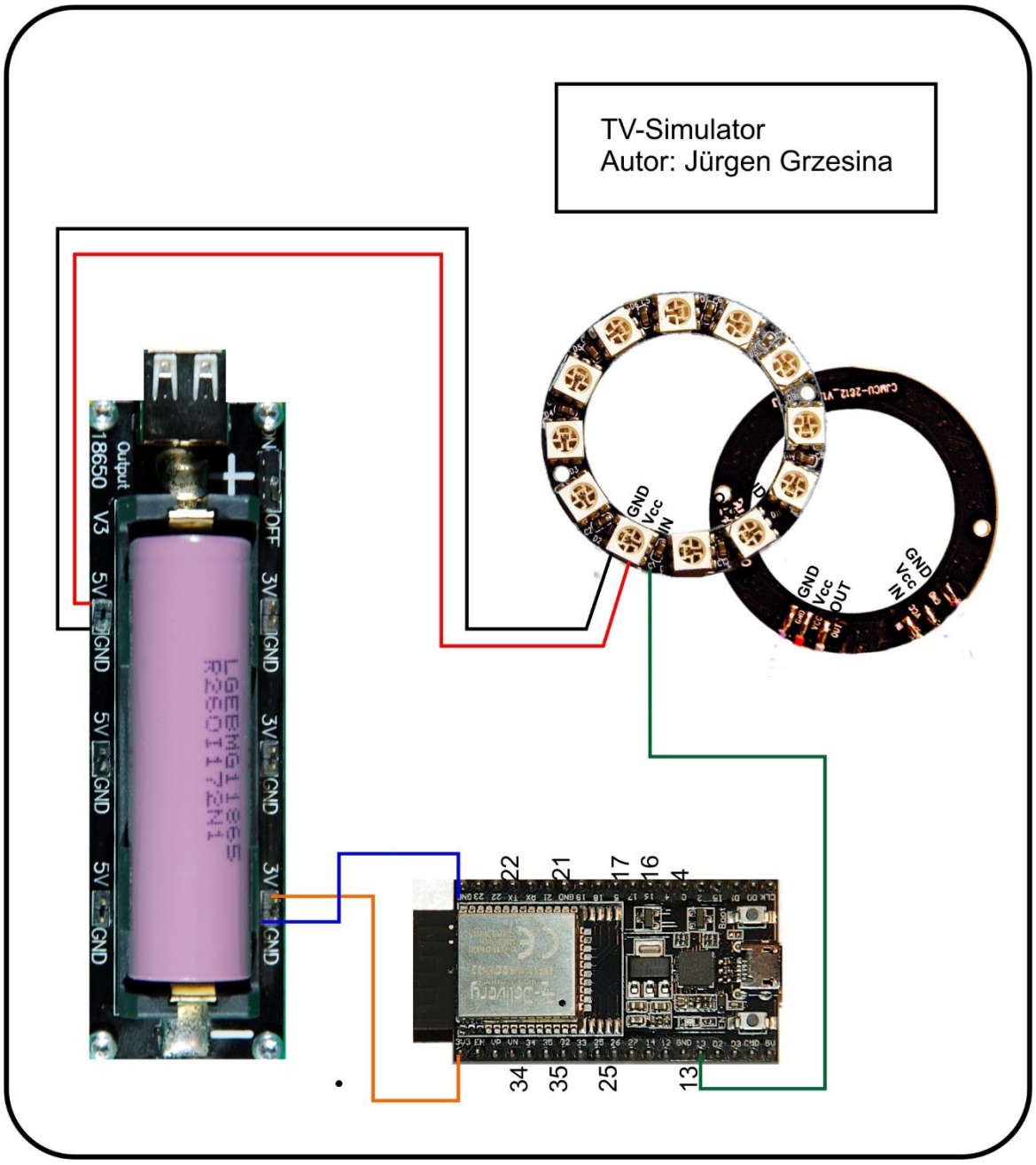

Abbildung 6: TVSIM-Schematic

Das Bemerkenswerte an diesem Projekt ist, dass es für das gesamte Spektrum an ESP-Prozessoren anwendbar ist. Fürs Flashen und die Programmierung eines ESP-01 ist aber ein entsprechender USB-Adapter nötig.

Hierfür verwende ich einen USB-TTL-Adapter mit einem Steckplatz für die Controllerplatine des ESP8266-01.

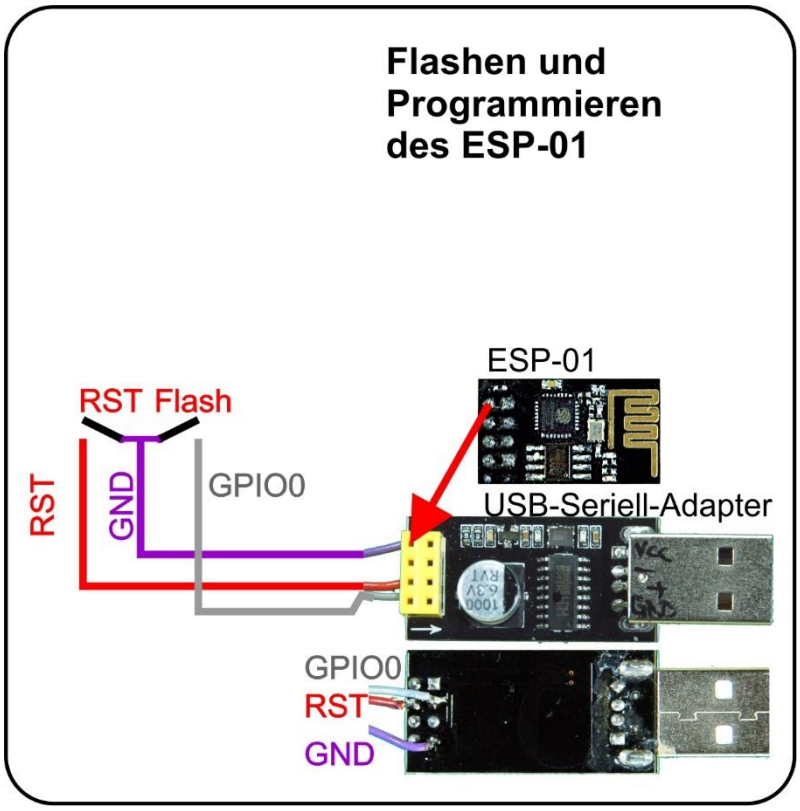

Abbildung 7: ESP-01-Beschaltung

In der Abbildung sehen Sie die drei Leitungen, die ich zusätzlich angebracht habe und die das Vorgehen deutlich erleichtern. Ich verbinde RST, GPIO0 und GND mit einem dreipoligen Kabel-Stück und löte daran eine Stiftleiste. so kann ich den Adapter an ein Breadboard stecken. Mittels Jumperkabeln lege ich dann den RST-Anschluss an einen Taster, der gegen GND schließt. Die GPIO0-Leitung kommt an einen weiteren Taster gegen GND. Nun kann ich die Pegelfolge, welche die Flashautomatik auf dem ESP32 oder auf vielen ESP8266-Boards programmgesteuert erledigt, manuell eingeben, RST + FLASH hold, RST release + FLASH hold, FLASH release. Denken Sie bitte daran, dass Sie beim ESP-01 den Vorgang zweimal durchführen müssen, einmal zum Löschen des Flashinhalts und ein zweites Mal zum Flashen der Firmware.

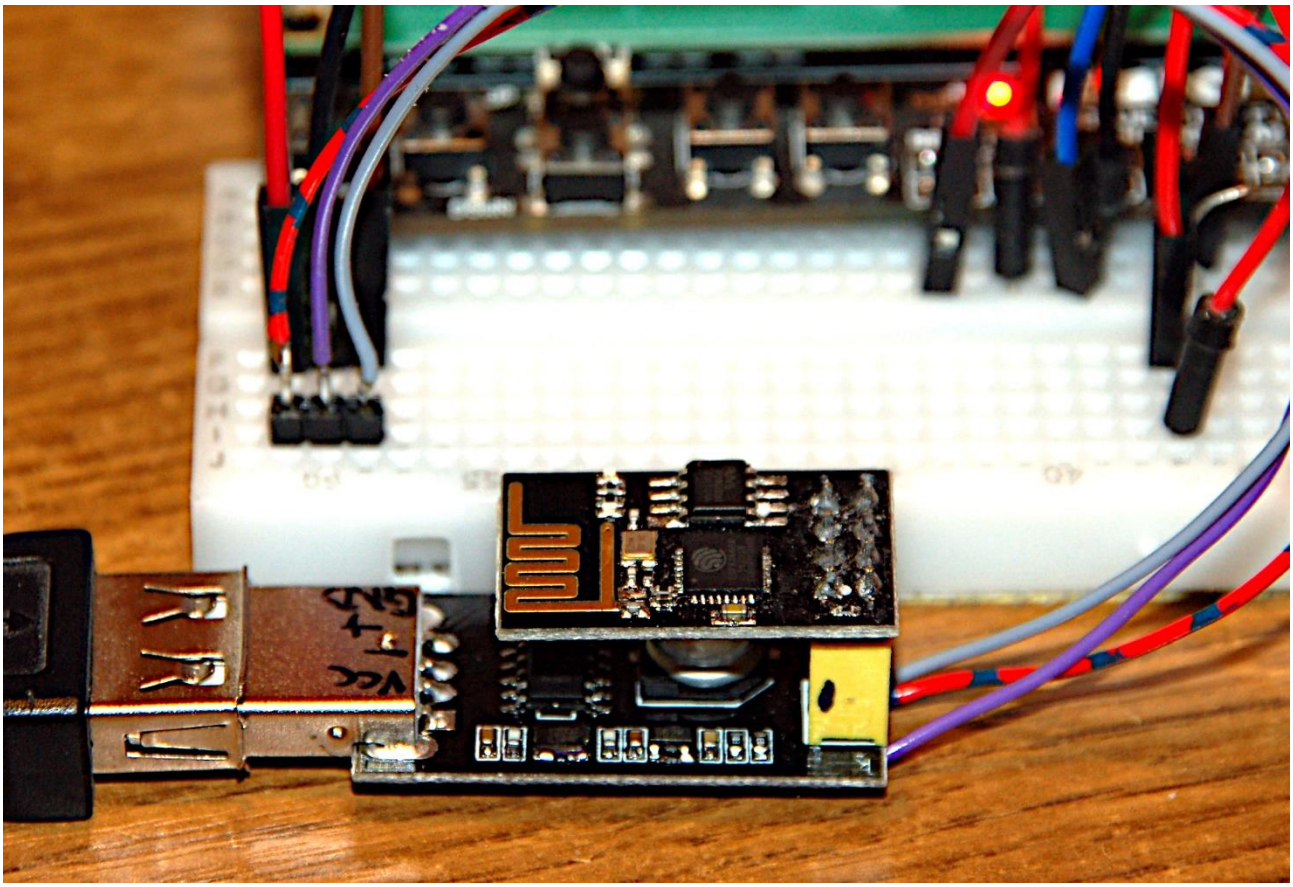

Abbildung 8: esp01-flashen

Das Procedere zum Brennen der Firmware ist ansonsten beim ESP32 und ESP8266-01 identisch. Laden Sie dazu die jeweils neuesten bin-Dateien herunter:
esp8266-1m-20210418-v1.15.bin
esp32-idf4-20210202-v1.14.bin

In Thonny öffnen Sie über **Run - Select interpreter** das Fenster **Thonny options**. Rechts unten öffnet ein Linksklick auf **Install or update firmware** den **Firmware installer**.

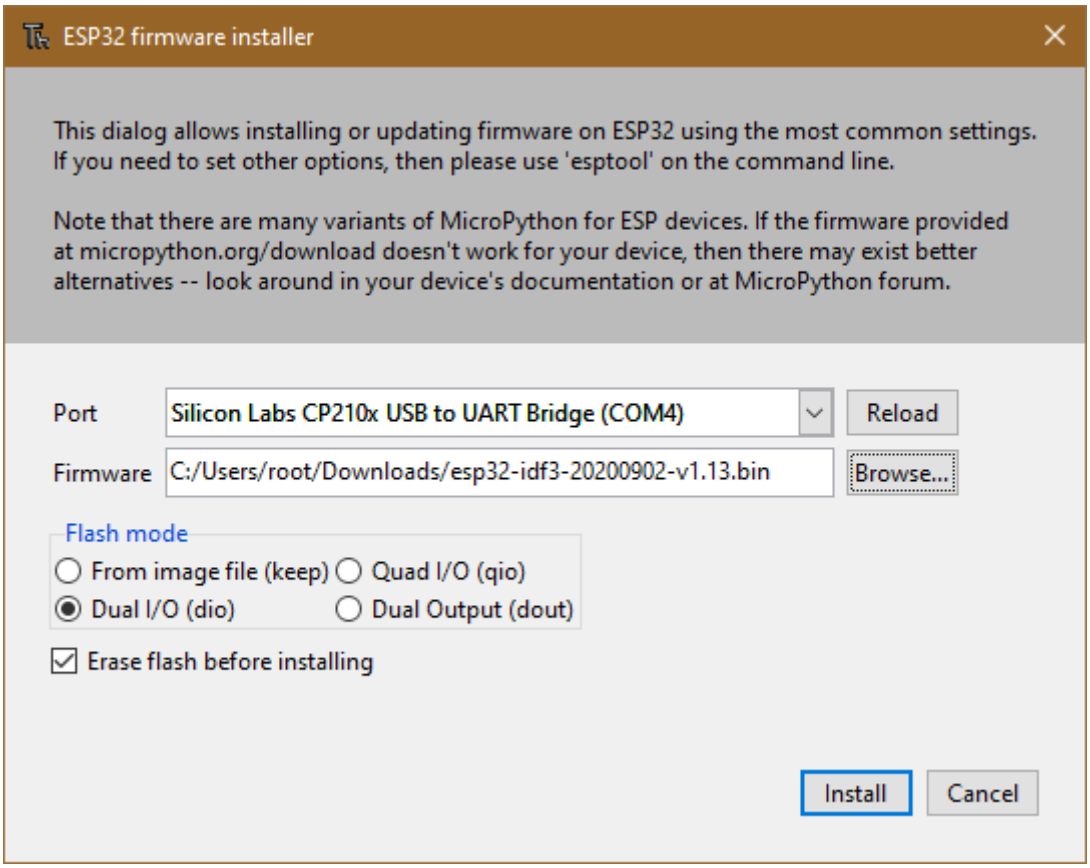

Abbildung 9: firmware_installer

Stellen Sie den Port ein und wählen Sie die entsprechende Firmwaredatei. Starten Sie nun den Brennprozess und denken Sie daran, beim ESP8266-01 die Tasten zu drücken.

Die Software

Verwendete Software:

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung. Darin gibt es auch eine Beschreibung wie die MicropythonFirmware auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die

MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, bevor der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm compilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen, über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Macro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein compiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer wie hier beschrieben.

Das Programm

Das Programm tvsim.py besteht aus zwei wesentlichen Teilen, der Klasse TVSIM und dem eigentlichen Hauptprogramm. Die Klasse TVSIM ist also dieses Mal in der Anwendung integriert und muss daher nicht importiert werden. Ferner wird nur der Scope einiger MicroPython-interner Klassen importiert.

Dennoch arbeitet das Programm recht trickreich. Hintergrund dafür ist der Umstand, dass verschiedene Leuchteffekte für die Farben rot, grün, blau und weiß parallel zueinander ablaufen müssen. Das ist mit Top-Down-Programmierung nicht zu machen. Folglich habe ich die vier Hardwaretimer der ESP-Chips ausgereizt. Sie fordern gezielt Programmunterbrechungen (aka Interrupt requests = IRQ) an. Dadurch werden die Effekte weitergeschaltet oder beendet. Weil die IRQs unabhängig voneinander erfolgen, sieht es so aus, als würden vier Programme zueinander parallel verlaufen. Damit die Effekte nicht periodisch auftreten, werden sie für die einzelnen Farben durch Zufallszahlen bestimmt. Gleiches gilt für die Zeitdauer der Effekte. Das Hauptprogramm sorgt nur dafür, dass ein neuer Effekt gestartet wird, sobald der vorhergehende abgelaufen ist. Die Herleitung der Parameter für den Start der Effekte, die von der Hauptschleife aus aktiviert werden, kann programmtechnisch verändert werden. Das beeinflusst die Ausprägung des Effekts von paranoid über nervös bis ausgeglichen. Ein weiterer Trick bewirkt die Verkürzung der Hauptschleife auf ein Viertel des eigentlich nötigen Umfangs.

Habe ich Sie neugierig gemacht, dann gehen wir jetzt ins Detail. Die Klasse TVSIM instanziert nicht das Verhalten für einzelne LEDs oder Durchlaufeffekte wie in Ringmaster1 und 2, sondern erzeugt Objekte für die Grundfarben rot, grün, blau und weiß. Jedes Farbobjekt verfügt über den gleichen Satz an Attributen und Methoden. Aber alle Farbobjekte steuern dasselbe NeoPixel-Objekt. Welche Neopixeleinheiten jeweils angesprochen werden, legen wir im Vorspann des Programms fest, ebenso die Grundhelligkeiten.

```
blue=(0,3,7)
green=(2,5,11)
red=(9,)
white=(1,4,6,8,10)

redStart=32
greenStart=10
blueStart=16
whiteStart=16
```

Der Konstruktor der Klasse TVSIM nimmt eine Reihe von Positionsparametern, die ein Farbobjekt erzeugen, dem die eben definierten Pixeleinheiten angehören. Dieser Ansatz wurde gewählt, weil damit jede Farbe über einen eigenen Timer sowie eigene Steuerattribute verfügt.

TVSIM(timer,neo,leds,color,startValue)

timer ordnet dem Objekt einen der 4 Hardwaretimer zu.

T0=Timer(0)
T1=Timer(1)
T2=Timer(2)
T3=Timer(3)

Alle Objekte sprechen jedoch dasselbe Neopixelobjekt np an.

from neopixel import NeoPixel
NeoPin=const(13)
NeoCnt=const(12)
np=NeoPixel(Pin(NeoPin),NeoCnt)

rot=TVSIM(T0,np,red,"red",redStart)
gruen=TVSIM(T1,np,green,"green",greenStart)
blau=TVSIM(T2,np,blue,"blue",blueStart)
weiss=TVSIM(T3,np,white,"white",whiteStart)

leds nimmt das Tupel mit den Nummern der Einheiten zur entsprechenden Farbe

dienste= 4
Farben=["rot","gruen","blau","weiss"]

color nimmt die Farbe im Klartext und bestimmt damit den Index aus der Liste Farben.

startValue legt die Grundhelligkeit der Farbe fest.

Die Farbinformation des Neopixel-Buffers np wird in Tupeln gespeichert. Tupel sind aber leider **immutabel**, das bedeutet, dass deren Elemente nicht verändert werden können. Weil wir genau das tun müssen, verwenden wir in den Methoden der Klasse TVSIM vorerst eine Liste für die Helligkeitswerte der Einzelfarben. Dann bauen wir daraus in zwei Stufen das Tupel zusammen, das dann durch die Methoden von TVSIM im Buffer np abgelegt wird. Wir kommen noch näher darauf zu sprechen.

Eine besondere Bedeutung kommt dem Instanzattribut **bussy** zu. Mit diesem Flag wird durch True signalsiert, dass gerade ein Effekt in Arbeit ist. Dann ist es nicht sinnvoll die Einstellungen des Effekts zu verändern. Das Hauptprogramm berücksichtigt das und startet erst dann einen neuen Effekt, wenn das bussy-Flag der entsprechenden Farbe den Wert False hat. Zusätzlich verzögern die Methoden zum Einleiten eines Effekts dessen Start solange, bis das bussy-Flag auf False steht.

Zur Initialisierung jeder Farbe muss beim Programmstart **initColor**() aufgerufen werden. Anhand dieser Methode lässt sich schön das Vorgehen beim Setzen eines Farbwerts ersehen. Der Farbwert ist eine Ganzzahl zwischen 0 und 255. Der Wert muss durch beliebige mathematische Operationen veränderbar sein. Wenn das geschehen ist, wird daraus das RGB-Tupel aufgebaut. Der **colorIndex** gibt Auskunft darüber, welche Farbe wir bearbeiten. Jede Methode aus TVSIM muss für jede Farbe taugen, aber sie muss auch wissen, für welche Farbe sie zuständig ist. Die Info bekommt jedes Farbobjekt im Konstruktor mit auf den Weg.

```
    def initColor(self):
        self.currentColorValue=self.basisWert
        self.oldColorValue=self.basisWert
        if self.colorIndex==3:
            r,g,b=self.basisWert,self.basisWert,self.basisWert
```

```
        else:
            self.components[self.colorIndex]=self.basisWert
            r,g,b=self.components
        self.color=r,g,b
        for n in self.pixNumbers:
            self.ring[n]=self.color
        self.ring.write()
```

Bei manchen Effekten ist es wichtig, zu wissen, welcher Farbwert vor einer Änderung aktiv war. Dafür gibt es die Merker **currentColorValue** und **oldColorValue**. Für weiß erhalten alle Farbkomponenten denselben Wert. Für rot, grün und blau bestimmt der colorIndex die Position des Werts in der Liste components, die im Konstruktor mit drei Nullen vorbelegt wurde. Wir entpacken die Liste in die einzelnen Komponenten r, g und b, die wir gleich danach zum Tupel color zusammenfügen. So kommt die Helligkeitsstufe an die richtige Stelle im Tupel **color**, während die anderen beiden auf 0 bleiben. In der for-Schleife weisen wir das Tupel allen beteiligten Positionen der Pixelgruppe zu. Mit **ring.write**() bringen wir die Pixel zum Leuchten. Dieses Vorgehen finden Sie in jedem der Effekte wieder. Unterschiedlich ist lediglich die Vorgabe oder Berechnung der Helligkeitswerte.

Und noch etwas kommt bei den Methoden für Effektstarts dazu. Sie alle haben am Schluss den Start des eigenen Timers mit der Zeitdauer aus dem Parameter **delay** gemeinsam. In diesem Aufruf ist auch die Routine als Callback-Funktion angegeben, die bei Ablauf des Timers aufgerufen werden soll.

```
    self.timer.init(mode=Timer.ONE_SHOT,period=\
                    self.duration, callback=self.ISRfadeOut)
```

Diese ISRs (Interrupt Service Routines) beenden, mehr oder weniger aufwändig, den Effekt, indem sie bussy = False setzen. Alle diese ISRs müssen neben dem Parameter self einen Parameter, hier heißt er tim, haben, der die Information zum Timer aufnimmt. Wird kein Parameter angegeben, gibt es eine Fehlermeldung.

Die Methoden fadeIn() und fadeOut() können und müssen sich selbst als callback angeben, denn das Fading geht ja über mehrere Helligkeitsstufen. Deren Wert wird bei jedem Durchlauf verändert, bis der Zielwert erreicht oder überschritten ist. Erst dann wird der Effekt durch Angabe der finalisierenden ISR beendet.

Das Hauptprogramm tut nichts anderes, als für alle Strings aus der Liste Farben das bussy-Flag zu überprüfen und, falls dessen Wert False ist, einen neuen Effekt mit neuen Parameterwerten zu würfeln. Damit das nicht für jede Farbe einzeln programmiert werden muss, verwende ich in der Liste **Farben** Strings. Die kann ich nämlich durch die Funktion **eval**()in Programmcode umwandeln lassen. So wird mit Color="rot" aus

```
eval(Color).bussy
eval(Color).setColor(farbWert,anzeigeZeit)
```

```
rot.bussy
rot.setColor(farbWert,anzeigeZeit)
```

Der Programmtext reduziert sich damit auf ein Viertel des sonst notwendigen Umfangs. Die generische for-Schleife iteriert hier nicht über einen Bereich von Zahlen, sondern über den Inhalt der Liste Farben. Diese Art for entspricht dem foreach in Perl.

Zum Abschluss kommt hier noch das gesamte Programmlisting.

```
# File: tvsim.py
# Author: Jürgen Grzesina
# Rev. 1.0
# Stand 21.06.2021
#
# ***************************************************
# Klasse TVSIM declarieren
# ***************************************************
class TVSIM:
    # TVSIM(Timerinstanz,NeoPixelInstanz,Pixelnummern-Tupel,
    #       Farbstring)
    def __init__(self,timer,neo,leds,color,startValue):
        self.mode=None
        self.ring=neo
        self.farbe=color
        self.currentColorValue=startValue
        self.oldColorValue=startValue
        self.pixNumbers=leds
        self.timer=timer
        self.colorIndex=["red","green","blue",\
                         "white"].index(color)
        self.basisWert=startValue
        self.color=(0,0,0)
        self.components=[0,0,0]
        self.name=color
        self.fadeStep=1
        self.bussy= False
        print("TVSIM: Farbe {} instanziert".format(color))

    def initColor(self):
        self.currentColorValue=self.basisWert
        self.oldColorValue=self.basisWert
        if self.colorIndex==3:
            r,g,b=self.basisWert,self.basisWert,self.basisWert
        else:
            self.components[self.colorIndex]=self.basisWert
            r,g,b=self.components
        self.color=r,g,b
        for n in self.pixNumbers:
            self.ring[n]=self.color
        self.ring.write()

    def setColor(self,col,delay):
```

```
        self.bussy=True
        self.oldColorValue=self.currentColorValue
        self.currentColorValue=col
        if self.colorIndex==3:
            r,g,b=col,col,col
        else:
            self.components[self.colorIndex]=col
            r,g,b=self.components
        self.color=r,g,b
        for n in self.pixNumbers:
            self.ring[n]=self.color
        self.ring.write()
        self.timer.init(mode=Timer.ONE_SHOT,period=delay,\
                        callback=self.ISRunblockColor)

    def clearColor(self,delay):
        self.bussy=True
        self.oldColorValue=self.currentColorValue
        self.newColorValue=0
        self.color=(0,0,0)
        for n in self.pixNumbers:
            self.ring[n]=self.color
        self.ring.write()
        self.timer.init(mode=Timer.ONE_SHOT,period=delay,\
                        callback=self.ISRunblockColor)


    def boostColor(self,col,duration):
        self.oldColorValue=self.currentColorValue
        self.currentColorValue=col
        self.bussy=True
        if self.colorIndex==3:
            r,g,b=col,col,col
        else:
            self.components[self.colorIndex]=col
            r,g,b=self.components
        self.color=r,g,b
        for n in self.pixNumbers:
            self.ring[n]=self.color
        self.ring.write()
        self.timer.init(mode=Timer.ONE_SHOT,period=duration,\
                        callback=self.ISRresetColor)

        # Tupel (linear|progressive, colorVal,colorVal,step,
        #        Dauer einer Phase)
        # if step >0 fadeIn step <0 fadeOut
    def setFading(self,mode,start,end,step,delay):
        while self.bussy:
            pass
        self.mode=mode
        self.start=start
        self.end=end
```

```
        self.step=step
        self.duration=delay
        if (step >0 and start>end) or (step <0 and start<end):
            self.end=start
            self.start=end
        self.oldColorValue=self.currentColorValue
        self.currentColorValue=self.start

    def getFading(self):
        return (self.mode,self.start,self.end,self.step,\
                self.duration)

    def startFading(self):
        while self.bussy:
            pass
        if self.step > 0:
            self.timer.init(mode=Timer.ONE_SHOT,period=\
                            self.duration, callback=\
                            self.ISRfadeIn)
        if self.step < 0:
            self.timer.init(mode=Timer.ONE_SHOT,period=\
                            self.duration, callback=\
                            self.ISRfadeOut)

    def getCurrentColorValue(self):
        return  self.currentColorValue

# ************************************************
# IRQ-Service-Routinen

    def ISRunblockColor(self,tim):
        self.bussy=False

    def ISRfadeIn(self,tim):
        col=self.currentColorValue
        self.currentColorValue += self.step
        #print(self.currentColorValue)
        if self.currentColorValue <= self.end:
            self.bussy=True
            if self.colorIndex==3:
                r,g,b=col,col,col
            else:
                self.components[self.colorIndex]=col
                r,g,b=self.components
            self.color=r,g,b
            for n in self.pixNumbers:
                self.ring[n]=self.color
            self.ring.write()
            self.timer.init(mode=Timer.ONE_SHOT,\
                            period= self.duration,\
                            callback=self.ISRfadeIn)
        else:
```

```
            self.oldColorValue=col-self.step
            self.timer.init(mode=Timer.ONE_SHOT,\
                            period=self.duration, \
                            callback=self.ISRresetColor)

    def ISRfadeOut(self,tim):
        col=self.currentColorValue
        self.currentColorValue += self.step
        #print(self.currentColorValue)
        if self.currentColorValue >= self.end:
            self.bussy=True
            if self.colorIndex==3:
                r,g,b=col,col,col
            else:
                self.components[self.colorIndex]=col
                r,g,b=self.components
            self.color=r,g,b
            for n in self.pixNumbers:
                self.ring[n]=self.color
            self.ring.write()
            self.timer.init(mode=Timer.ONE_SHOT,period=\
                            self.duration, callback=\
                            self.ISRfadeOut)
        else:
            self.oldColorValue=col-self.step
            self.timer.init(mode=Timer.ONE_SHOT,\
                            period=self.duration, \
                            callback=self.ISRresetColor)

    def ISRresetColor(self,tim):
        self.currentColorValue=self.oldColorValue
        if self.colorIndex==3:
            r,g,b=self.currentColorValue,\
                  self.currentColorValue,\
                  self.currentColorValue
        else:
            self.components[self.colorIndex]=\
                            self.currentColorValue
            r,g,b=self.components
        self.color=r,g,b
        for n in self.pixNumbers:
            self.ring[n]=self.color
        self.ring.write()
        self.bussy=False

# Ende Klassendefinition TVSIM
# ************************************************

# ************************************************
# Importgeschaeft
# ************************************************
import os,sys       # System- und Dateianweisungen
```

```
import esp          # nervige Systemmeldungen aus
esp.osdebug(None)

import gc           # Platz fuer Variablen schaffen
gc.collect()
#
from machine import Pin,I2C,Timer
from time import sleep
from neopixel import NeoPixel

NeoPin=const(13)
NeoCnt=const(12)

np=NeoPixel(Pin(NeoPin),NeoCnt)

blue=(0,3,7)
green=(2,5,11)
red=(9,)
white=(1,4,6,8,10)

redStart=32
greenStart=10
blueStart=16
whiteStart=16


# *************** Funktionen declarieren ******************
def clearRing():
    for i in range(NeoCnt):
        np[i]=(0,0,0)
    np.write()



# *************** Hauptprogramm *************************

clearRing()
T0=Timer(0)
T1=Timer(1)
T2=Timer(2)
T3=Timer(3)
rot=TVSIM(T0,np,red,"red",redStart)
gruen=TVSIM(T1,np,green,"green",greenStart)
blau=TVSIM(T2,np,blue,"blue",blueStart)
weiss=TVSIM(T3,np,white,"white",whiteStart)
dienste= 4
Farben=["rot","gruen","blau","weiss"]

while 1:
    for Color in Farben:
        if not eval(Color).bussy:
```

```
            service =(os.urandom(3)[2])%dienste
            if service==0: # setColor()
                params=os.urandom(3)
                farbWert=params[2]%100
                anzeigeZeit=(params[1]%64+64)*(params[0]%16)
                if anzeigeZeit==0:
                    anzeigeZeit=300
                print("{}.setColor({},{})".format\
                       (Color,farbWert,anzeigeZeit))
                eval(Color).setColor(farbWert,anzeigeZeit)
            if service==5: # clearColor()
                params=os.urandom(2)
                #anzeigeZeit=params[1]*(params[0]%6)
                anzeigeZeit=(params[1]%64+64)*(params[0]%16)
                if anzeigeZeit==0:
                    anzeigeZeit=300
                print("{}.clearColor({})".format\
                       (Color,anzeigeZeit))
                eval(Color).clearColor(anzeigeZeit)
            if service==1: # boostColor()
                params=os.urandom(3)
                farbWert=params[2]%128+128
                #anzeigeZeit=params[1]*(params[0]%6)
                anzeigeZeit=(params[1]%64+64)*(params[0]%16)
                if anzeigeZeit==0:
                    anzeigeZeit=300
                print("{}.boostColor({},{})".format\
                       (Color,farbWert,anzeigeZeit))
                eval(Color).boostColor(farbWert,anzeigeZeit)
            if service==3: # boostColor()
                params=os.urandom(5)
                modus=0
                startCol=params[2]%128+128
                endCol=params[3]%16
                schritt=(params[4]%20)-10
                #dauer=params[1]*(params[0]%6)
                dauer=(params[1]%128+128)*((params[0]%16)+16)
                if dauer==0:
                    dauer=300
                print("{}.startFading({},{},{},{})".format\
                       (Color,startCol,endCol,schritt,dauer))
                eval(Color).setFading(modus,startCol,\
                                      endCol,schritt,dauer)
                eval(Color).startFading()
```

Als Anwendung für TVSIM kommt ein Stand-Alone-Gerät ebenso in Frage, wie die Anbindung an oder auch Integration in den Attendance Guy. In letzterem Fall wäre dann sogar das Ein- und Ausschalten via SMS möglich.

Viel Freude und Erfolg beim Basteln und Tunen.

PDF in deutsch
PDF in english